

The Definition Layer of Piper

Brad Chapman (chapmanb@arches.uga.edu)

Last Update–5 June 00

Contents

1 Overview of the Definition Layer	1
1.1 Implementation	1
1.2 Functions of the DL	2
1.3 Communication Protocols	2
1.3.1 User Interface to Definition Layer	2
1.3.2 Definition Layer to Processing Layer	2
1.3.3 Remote Computer Location and Authentication	3
1.3.4 Definition Layer to Definition Layer (remote communication)	3
2 Some HowTos for doing things in the DL	5
2.1 Storing Passwords within Piper	5
3 DL Internals–For people who care about the code.	5
3.1 Rationale	5
3.2 The structure of the DL classes	6
3.3 Short descriptions of DL classes	6
3.3.1 Classes which represent the workspace	6
3.3.2 Speciality Classes	7
3.3.3 Classes which make up the DL internals	7

1 Overview of the Definition Layer

The Definition Layer (DL) provides the glue for connecting two different parts of Piper:

The Build-Time System This part of Piper involves the user acting through the user interface to create an interconnecting set of programs to be executed.

The Run-Time System This part of Piper is responsible for execution of programs and piping information between programs.

The DL rests firmly between the User Interface Layer (UIL) and the brokering layer (BL) and mediates communication between these two sections of Piper. In addition, the DL is responsible for storing XML describing the current status of the userinterface, storing specialized XML descriptions of programs and libraries, and communicating with remote Piper implementations.

1.1 Implementation

The definition layer is implemented in the python programming language and relies on freely available XML and DOM libraries. CORBA communication will occur through omniORB <http://www.uk.research.att.com/omniORB/> using the omniORBpy bindings (<http://www.uk.research.att.com/omniORB/omniORBpy/>).

1.2 Functions of the DL

The main functions of the definition layer are:

1. Maintain communication with multiple user interfaces through a streaming XML dialog communication protocol. This communication occurs through a local socket and allows user interfaces to be developed in any programming language that provides support for sockets.
2. Storing an XML representation of the user interface. This XML representation models the connections between different nodes in the user interface and the interrelationships between them. These XML representations will allow users to save entire user interfaces or individual nodes, and will provide the ability for recovery after crashes.
3. Maintaining a permanent storage location for specially designed nodes that represent specific programs and libraries. The information for these nodes will be stored as XML and be available directly available to the user interface through the streaming XML dialog.
4. Converting the XML representation generated by the build-time system into an XML format that can be processed by the run-time system.
5. Communication with the processing portion of the program by implementing a CORBA client to a server in the brokering layer. This communication allows work flow diagrams to be passed to the processing layers for implementation, and allows processes to be queried while running.
6. Communication with remote Piper programs as both a client and a server using CORBA (through an idl that still has to be defined). This allows a specific implementation of Piper to serve out a specialized node or set of nodes to be utilized by other Piper implementations at remote locations. All aspects of this remote communication will pass through the definition layer.

1.3 Communication Protocols

1.3.1 User Interface to Definition Layer

Multiple user interfaces can communicate with the definition layer through a CORBA interface defined in the IDL (interface definition language) file `ui12dl.idl`. Further information on this communication is available in *How To Write a User Interface for Piper*. Using CORBA for this communication protocol allows user interfaces to be written in any language supporting CORBA language bindings.

1.3.2 Definition Layer to Processing Layer

Communication between the definition and processing layers occurs using a CORBA interface described in the interface definition language file `vsh_dl2bl.idl`. This communication occurs in three broad steps:

1. The definition layer negotiates a connection with the processing layer and receives a unique id, the `dlid` for which to identify itself in future communication. This connection is established by calling the following idl defined function:

```
interface Connection {
    processorS setup(in dlIdT dlid, in passwordT password,
        in connectionE connecttype) raises (IdInUse);
};
```

2. The definition layer uploads an XML document representing the programs to be processed and the connections between them. This XML document will then be parsed by the brokering layer and used to begin running the programs. The brokering layer returns a URI id which the definition layer can then use to further communicate with the brokering layer about the status of the processing. The upload occurs through the following method:

```

interface Representation {
    uriS upload(in string xmldocument, in dlIdT dlid,
               in passwordT password) raises (UriUndefined);
};

```

3. The definition layer then queries the brokering layer to determine the status of different programs. When programs that return results are finished, the definition layer will then retrieve these results and return them to the user interface to be displayed. Query and retrieval of processes are obtained through the following:

```

interface Obtain {
    statusS uriStatus(in uriS uri, in dlIdT dlid,
                     in passwordT password) raises (UriNotFound);

    string uriInfo(in passwordT processid, in uriS uri)
        raises (InfoNotAvailable);
};

```

1.3.3 Remote Computer Location and Authentication

The definition layer is also responsible for allowing two remote Piper implementations to connect and authenticate with each other. This location and authentication is a prerequisite for dl to dl communication (which is described in detail below).

The details of how this remote connection will work are still rather sketchy, but I'll outline what I know about this. Basically, the idea is that remote connection will be established using standard corba services, specifically the naming service to start with (and hopefully with the trading service added on later). I'll illustrate the basic way this will work using the naming service.

Figure 1 is a diagram of the basic setup, where Brad's Piper instance wants to connect to Jeff's remote Piper program. The connection is mediated by a separate "central" server running Piper with the naming service going. The reference to this "central" server will be well known (ie. available as a url or something) so that Piper programs can easily find it. This computer's job is to store object references to remote Piper implementations and associate them in a way that is easy to find. So in the example I just have a few computers associated by location (we could choose any association scheme we think will be easy). The idea is that every time a Piper instance starts, it registers with this remote computer so other computers can find it.

In our example, we will assume that Brad's computer wants to find Jeff's.

1. When Jeff's computer starts up it registers itself with the central naming service computer, so that it is available to be found (under Boston).
2. Brad's computer connects with the name server, and traverses the name server graph (first to 'Available Piper computers', then to 'Boston', then finally to 'Jeff'). Brad's computer then retrieves an object reference to Jeff's computer which allows it to connect to it.
3. Brad's computer sends an authentication message to Jeff's computer (for instance, if Brad's computer just wanted to use Jeff's public nodes, it would authenticate with the username 'public' and the password 'public.') Jeff's computer then accepts the authentication and returns an object reference allowing Brad's computer to do further queries (about available programs, etc). Then off we go to begin dl to dl communication. . .

1.3.4 Definition Layer to Definition Layer (remote communication)

The definition layer is also responsible for mediating communication between Piper implementations at different locations. This intercommunication occurs through a four step process. To describe this, let's look at figure 2, illustrating communication between two Piper implementations.

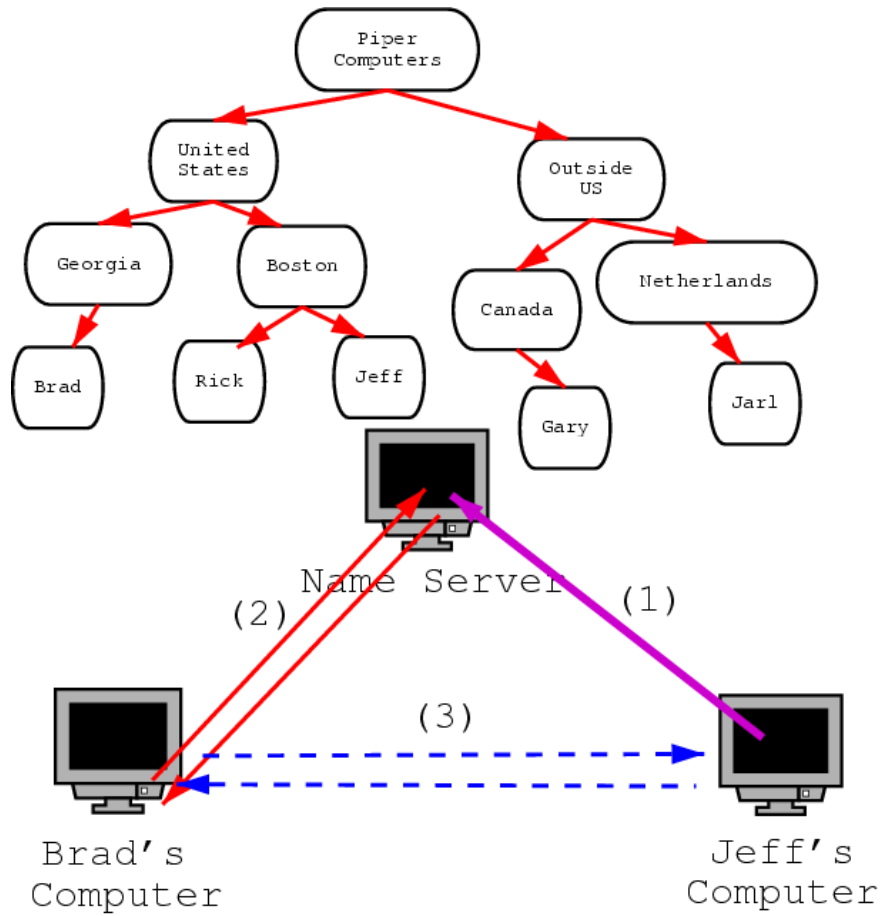


Figure 1: Locating Piper programs running remotely.

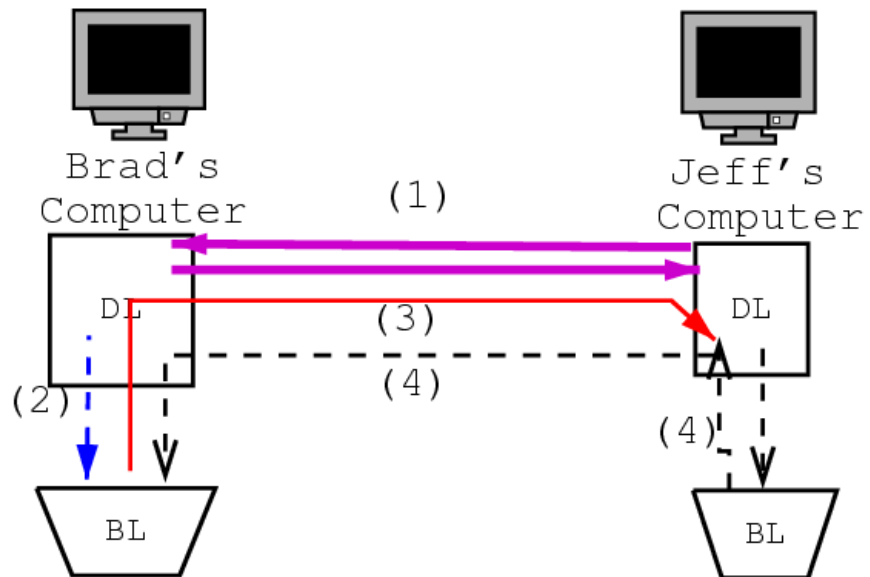


Figure 2: Communication between two remote piper programs

Now, let's imagine that Brad wants to run a really complex program, but his computer is so incredibly slow that it would take three years to run the thing, and the computer would probably end up exploding before it finished. However, Jeff, who is running his Piper system on a huge cluster of Sun workstations, is kind enough to let Brad run the program on his Piper implementation.

First, Brad's computer must locate Jeff's computer and then authenticate with it through the process described above. Once the computers are connected and ready to talk, the interchange proceeds as follows:

1. The definition layer (DL) on Brad's computer queries the DL on Jeff's computer and receives back information on what programs are available to be run there. This information is then available to Brad, manipulating the gui, who incorporates one of the programs on Jeff's computer into his work-flow diagram.
2. Brad submits the workflow diagram (in XML) to the Brokering Layer (BL) so that it can be processed. As the BL is dealing with running the applications and processing everything, it realizes that one of the nodes is located on Jeff's computer and is not local.
3. The BL on Brad's computer then sends a request to the DL on Brad's computer to go fetch the results from running this node on Jeff's computer. Brad's DL then sends the XML it needs processed to Jeff's DL.
4. Jeff's DL submits the XML for processing to the BL. This BL runs the program and gets the results, and then realizes they are needed remotely, and sends them back to the DL. Jeff's DL then returns them to Brad's DL, which submits them back to the BL which has been patiently waiting for them and ta-da, the process has been run remotely!

2 Some HowTos for doing things in the DL

2.1 Storing Passwords within Piper

Password storage has proven to be a tricky thing because we have a unique case. We'd need to be able to store passwords in a format this is retrievable back into plain text, but without having to store an unencrypted key somewhere in the filesystem. The way I finally settled on doing this is to have a password database which stores all of the passwords for a particular username and password. So to take care of your passwords you need to do the following:

1. Run the 'AddLociPass.py' script to create a specified username and password to store you passwords under.
2. Continue running 'AddLociPass.py' and enter passwords for all of the items you will need to access. For right now, the only thing we are using it for is for storing passwords for accessing databases (such as MySQL).
3. Once you've entered all of your passwords into the database, you are ready to run Piper. When the user interface connects with the definition layer, you will need to enter the username and password to enter the database and voila, you have access to your passwords during your session.

Please note that right now this isn't set up completely yet (since we don't have a user interface for entering usernames and passwords for authentication), but this is how I am planning it to work in the future, so you know. For testing purposes, the ui now authenticates with the the username 'default' and the password 'default', so if you create a (non-secure) password database with this username and password, you will have access to your passwords through this system.

3 DL Internals—For people who care about the code.

3.1 Rationale

When I started trying to combine Loci with GMS I rapidly realized that I had done an extremely poor job of designing the relationship between classes in the definition layer, and began to feel thoroughly ashamed of myself. In addition, I realized that there were a number of cheap tricks I was using to get around the fact that the organization was very poor, and that I couldn't get away from these without a redesign. So, anyways, I have tried to redesign the code to

eliminate my cheap tricks (which were all bad security holes :-)) and to take advantage of the object oriented nature of python (which I hadn't been doing well enough before). This is my attempt to document the structure of the definition layer so people can understand it and offer suggestions and criticisms for further improvement.

3.2 The structure of the DL classes

Figure 3 illustrates the overall structure of the Definition Layer classes (Note: I don't really know anything about UML, but just tried to use it as best as I could in Dia. So if there are lots of confusing errors in this, please let me know so I can fix them.)

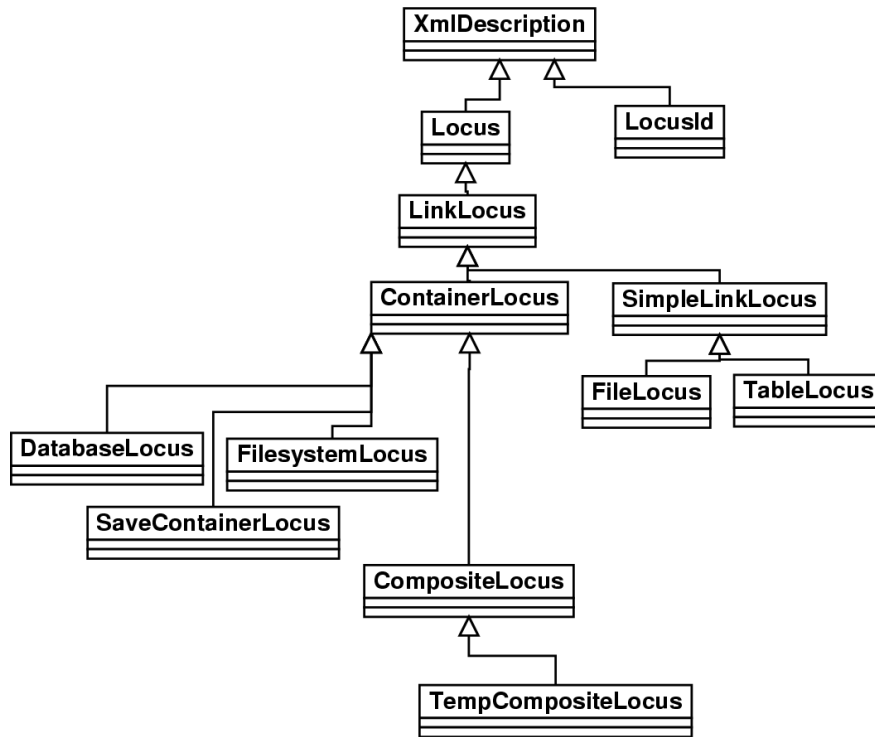


Figure 3: The overall structure of the definition layer classes

3.3 Short descriptions of DL classes

3.3.1 Classes which represent the workspace

XmlDescription This is the base class from which all other workspace representation derive. This class provides functions for dealing directly with XML files saved in the workspace. The idea is that this class is the only one that would need to be significantly rewritten if we were to switch the storage medium for XML from the filesystem to a database.

Locus This is the basic Locus class and just provides functions which will be useful in all different kinds of locus types.

LinkLocus This is the first “useful” locus type since it defines a locus which can initiate connections with other loci.

ContainerLocus This represents a locus that is able to hold other loci.

CompositeLocus This represents a locus that is not only able to hold loci, but can also do things with them such as create loci *de novo*, and connect two loci.

3.3.2 Speciality Classes

FileLocus A locus type for representing an individual file.

FilesystemLocus A locus type for representing a local filesystem directory.

TableLocus A locus type for representing an individual table in a database.

DatabaseLocus A locus type for representing a whole database.

SaveContainerLocus A locus type representing a directory which holds permanent storage loci.

TempContainerLocus Represent the directory holding loci which are temporary and will only be stored while the program is running.

3.3.3 Classes which make up the DL internals

LocusId This class provides facilities for storing locus ids in association with the actual position of a locus in the filesystem. This way, Piper can pass 10 digit ids around to represent different loci, but the DL can get a hold of actual XML files by finding the file associated with the id from LocusId.

BaseServer A CORBA server implementing the `DlServer` interface. This server handles requests for authentication from user interfaces, and removes user interfaces when they have died.

DI* in uicommunication.py These are all classes for handling particular requests from the definition layer. They implement the CORBA interface classes allowing a user interface to interact with the definition layer.

DIMain This is the main definition layer class which will start up and manage all the subprocesses that need to run within the DL.

LociPass This provides a simple password database for storing passwords securely within Piper.